# Parallel Execution of ATL Transformation Rules

Massimo Tisi, Salvador Martínez and Hassene Choura

AtlanMod, École des Mines de Nantes - INRIA, LINA, Nantes, France
firstname.lastname@inria.fr

**Abstract.** Industrial environments that make use of Model-Driven Engineering (MDE) are starting to see the appearance of very large models, made by millions of elements. Such models are produced automatically (e.g., by reverse engineering complex systems) or manually by a large number of users (e.g., from social networks). The success of MDE in these application scenarios strongly depends on the scalability of model manipulation tools. While parallelization is one of the traditional ways of making computation systems scalable, developing parallel model transformations in a general-purpose language is a complex and error-prone task. In this paper we show that rule-based languages like ATL have strong parallelization properties. Transformations can be developed without taking into account concurrency concerns, and a transformation engine can automatically parallelize execution. We describe the implementation of a parallel transformation engine for the current version of the ATL language and experimentally evaluate the consequent gain in scalability.

## 1 Introduction

Part of the industrial landscape looks at tools based on Model-Driven Engineering (MDE) to handle in a uniform way a plethora of software engineering tasks at development/maintenance time. Some examples are the development of critical systems [8], reverse engineering and modernization [20], artifact management [5]. MDE tools are also used at runtime, in systems built around the manipulation of model-based abstractions during system execution [6]. The most popular MDE frameworks, like the Eclipse Modeling Framework (EMF), are inspired by the OMG's Meta-Object Facility (MOF), and provide facilities to define and manipulate metamodels and conforming models. Model manipulation operations on these frameworks can be developed using APIs in general-purpose languages (the most popular approach) or by specific model-transformation languages (MTLs) originally designed to ease development, analysis and maintenance of the model-manipulation code.

Some of the companies that embraced (or want to embrace) MDE need to handle huge amounts of data. In MDE terms this reflects in the need to manipulate very large models (VLMs), e.g. models made by millions of model elements. Examples of such models appear in a wide range of domains as shown in industrial cases provided by literature: in [4] the authors work over industrial AUTOSAR[2] models with over 1 million model elements; [18] analyses civil-engineering related models with more that 7 million computational objects; in

the area of model-driven software product lines, [17] handles product families with up to 10 million model elements. Reverse engineering tasks may also produce large models as shown in [3], where our team obtains large models with up to 5 million model elements from the Eclipse JDT sources.

Due to the physical constraints preventing frequency scaling in modern CPUs, multi-core architectures are very popular today, making parallelism a cost-effective solution to improve computation times for VLMs. However, using a general-purpose language, parallel programs are more difficult to write with respect to sequential programs [16], mainly because of: 1) new classes of potential bugs introduced by concurrency (e.g., race conditions); 2) the difficulty in getting good parallelization while handling communication and synchronization between the different concurrent tasks; 3) increased difficulty in debugging the parallel code. One of the well-known approaches to simplify parallel programming is relying on implicitly parallel languages, and several such languages are available [1]. Using implicit parallelism, the developer does not need to worry about dividing the computation and handling communication and synchronization. The language implementation takes care of these aspects and the development of parallel programs is substantially simplified, which results in a significant productivity gain.

In this paper we want to show that ATL (the AtlanMod Transformation Language [14]), a rule-based model-transformation language designed with the principle of rule independence can be overloaded with implicit parallelism. By running on a parallelized engine, the execution time of ATL model transformations can scale well on the number of processors. While implicit parallelism had a limited success in general-purpose languages, we argue that the specific task of model transformation on VLMs may greatly benefit from it.

We provide the following contributions:

- We study the parallelization of the ATL language, separating it in two indipendent problems of transformation language parallelization and query language parallelization and we address the first analyzing decomposition and synchronization aspects.
- We provide a multi-threaded implementation of the ATL engine by adapting the standard engine. The resulting compiler and virtual machine are publicly available[2] and we plan to merge them in the next default version of ATL.
- We experimentally measure the improvement in scalability, by comparing the execution times of the same transformation in three semantically equivalent implementations: 1) a Java implementation, 2) an ATL implementation running on the standard engine, 3) the same ATL implementation running on the multi-threaded engine. Since no other change is performed to ATL, this experimentation gives an idea of the net effect of the parallelization.

In this paper we apply our approach to the development of a multi-threaded version of ATL with the aim of improving scalability on multi-core and multi-processor computers. However we plan in future work to adapt our automatic

[1] `http://en.wikipedia.org/wiki/Implicit_parallelism`

[2] `http://www.emn.fr/z-info/atlanmod/index.php/Parallel_ATL`, EPL licence

parallelization approach and apply it to distributed environments, with the aim to implement a distributed engine for ATL.

The paper is structured in the following way: Section 2 introduces the ATL transformation language and the running case whereas Section 3 details the parallelization problem and the proposed approach. Section 4 describes the implementation of the parallel engine for ATL and Section 5 presents the results of its performance evaluation. Section 6 discusses related work before the final Section 7 that summarizes conclusions and future works.

## 2 The ATL Language

To briefly illustrate the ATL language we rely on a small example on which we base also the experimentation section, i.e. the Class2Relational transformation[3] that transforms class diagrams into relational models. In Listing 1.1 we show an excerpt of this transformation (the full code can be found on the paper's website) and Fig. 1 illustrates its application to a very small model.

**Listing 1.1.** ATL Class2Relational transformation (excerpt).

```
1  rule Class2Table {
2    from
3      c : ClassDiagram!Class
4    to
5      out : Relational!Table (
6        name <- c.name,
7        cols <- c.attr->select(e | not e.multiValued),
8        key <- Set {key}
9      ),
10     key : Relational!Column (
11       name <- 'objectId',
12       type <- thisModule.objectIdType()
13     )
14 }
15
16 rule DataTypeAttribute2Column {
17   from
18     a : ClassDiagram!Attribute (
19       a.type.oclIsKindOf(ClassDiagram!DataType) and not a.multiValued
20     )
21   to
22     out : Relational!Column (
23       name <- a.name,
24       type <- a.type
25     )
26 }
```

The listing shows two rules, respectively responsible of transforming Classes into Tables with their respective Key column (Class2Table) and single-valued primitive-type Attributes into Columns (DataTypeAttribute2Column). Rules transform occurrences of the *input pattern (from)* in occurrences of the *output pattern (to)*. Occurrences of the input pattern may be filtered by introducing a *guard*, a boolean condition that source model elements must satisfy (e.g. line 19). Elements of the output pattern can have their features initialized through the use of *bindings*, expressions computing the values to assign to each feature (lines 6-8, 11-12, 23-24). For expressing guards and bindings, ATL relies on a separate *query language*, the OMG's Object Constraint Language (OCL). To help in factorizing OCL code ATL allows the definition of OCL functions, named

[3] http://www.eclipse.org/atl/atlTransformations/#Class2Relational

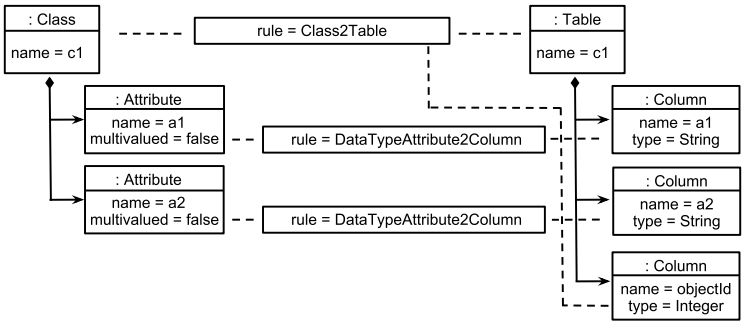


**Fig. 1.** Application of Class2Relational to a minimal model. For each rule application a trace link is generated labeled with the name of the applied rule.

*helpers*. For instance the binding at line 16 calls the objectIdType() helper (not included in the excerpt) that looks in the source model for a usable datatype for identifiers. The two rules in the Listing 1.1 are examples of *matched* rules, declarative rules that are spontaneously triggered when they match elements in the source model. The language allows also the definition of *lazy* rules, that need to be explicitly triggered by other rules.

Two features of the ATL language are not considered in the rest of the paper and are left to future work: 1) ATL includes an imperative part that does not increase the expressive power of the language, but is designed to simplify the implementation in complex cases, that would be verbose to encode in declarative rules; 2) ATL includes a separate execution mode called refining mode, in which transformation rules are applied in-place for the refinement of the source model. In the following we implicitly refer to the declarative part of ATL in standard execution mode.

The standard execution mode of ATL imposes a few constraints that result important for parallelization, since they strongly limit the possible dependencies between rules:

- During the matching phase, output elements of transformation rules are immediately instantiated and added to the target model, and are not subject to following matches. This means that the output of a rule cannot be used as intermediate data and cannot be transformed or deleted by another rule[4]. This constraint is one of the main differences between ATL and typical MTLs based on graph transformations.
- OCL expressions are never allowed to navigate the target model. The consequence is that the OCL expression that calculates a binding cannot use the output of another rule.
- Single-valued properties in the target model are assigned once and are not updated again during the transformation execution.

[4] Lazy rules can be triggered recursively, but they always match over the source model and not over the target elements previously generated.

– Multi-valued properties in the target model can be updated multiple times, but only for adding new values (this allows for incremental construction of the property).

Because of these constraints matched rules depend on each other only in one case, i.e. when they generate elements connected by a reference. In the example the rule Class2Table generates a Table whose reference *cols* has to be connected to some of the columns generated by the rule DataTypeAttribute2Column. The connection (line 7) is made by calculating the set of Attributes in the source model that correspond to the Columns to connect (in the example all the attributes of the matched Class that are not multivalued). ATL will implicitely fill the *cols* reference with all the Columns that are generated by any rule matching the Attributes calculated at line 7. This mechanism is called *implicit resolution algorithm*.

ATL transformations are compiled in a bytecode format called ASM, interpreted by the ASM virtual machine. The ATL architecture together with the full execution algorithm for ATL transformations is described in [14].

## 3 ATL Parallelization

As several other transformation languages, the ATL language embeds a separate query language that allows to define expressions over the models under transformation. In the ATL case the query language is the functional language OCL, and ATL restricts its use on computations over the source model. This constraint makes the execution of ATL and OCL two independent phases: 1) ATL launches the execution of OCL code from guards or bindings; 2) OCL calculates a result in a side-effect free way, by navigating the source model and possibly calling other OCL functions (helpers), and returns the result to ATL.

This separation makes the two problems of parallelizing ATL and OCL completely independent. The automatic parallelization of OCL code is a typical problem of parallelization of a functional language, and it is already studied in literature [21]. For this reason in the following we will deal only with the parallelization of the rule execution language. Our resulting engine will of course support OCL expressions but they will be executed in the same task of the rule application that launches them. A parallel engine for OCL may be integrated in future, and it will not require changes to the parallelization mechanism discussed in this paper.

In the next two sections we consider the parallelization of the transformation execution language as a problem composed by two orthogonal subproblems.

1. Decomposition, i.e., how to decompose the transformation computation to parallelize the calculation.
2. Synchronization, i.e., how to coordinate the dispatched tasks and manage their inter-communication. As we are using shared memory structures, concurrent access to these data structures has to be optimized to maximize parallelism.

### 3.1 Decomposition

The computation of a model transformation is composed of 1) a set of expression evaluation over source model elements (matchings), 2) a set of rule applications, one for each match found in the first set. A significant part of the computation of each rule application seems to be independent from other rule applications, suggesting the possibility of executing each rule application and each match in a different thread. While the approach would be probably suitable to small models, VLMs would force the engine to instantiate millions of tasks per transformation. Even with the support of an efficient job scheduler, responsible of assigning the jobs to a limited fixed set of threads, the cost of instantiating, keeping in memory and synchronizing between millions of jobs would overtake the benefits of parallelism.

For this reason we look to a more coarse-grained decomposition for the transformation computation. Traditional literature on parallelism distinguishes two opposite approaches (and a set of intermediates between the two): task parallelism and data parallelism.

**Task parallelism.** In task parallelism, each task contains a different set of operations, but works on the same data set. The approach is especially convenient when the fact of working on the same data does not introduce dependencies among the execution threads.

In our model transformation scenario, an example of task parallelism is grouping the computation *by rule* so that: 1) each task executes a different rule, including the OCL expressions for guards and bindings; 2) each task works on the full source and target models.

In this paper we will follow this approach, motivated by our main argument: the ATL language structures the computation in rules, that the language constraints (see Section 2) make highly independent from each other. As we will see in the next section, the manipulation of shared data will introduce synchronization issues that we will need to address relying on the ATL specificities.

After dividing the computation by rule, we have the option to further decompose the rule in two execution threads for the two well-defined phases of a rule execution: matching and rule application. Since every rule application needs to rely on the output of its matching phase, the strong dependency between the two threads hampers a direct improvement in parallelization. However, as we will see in the next section, dividing matching and rule application provides a better flexibility that we can exploit for improving synchronization.

In summary we instantiate two jobs for rule. For instance, the execution of the limited excerpt in Listing 1.1 results in four jobs:

- a *match* job for Class2Table looks for elements of type Class, and for each one it instantiates a trace Link, together with an empty Table and an empty Column as placeholders for the next job.
- an *apply* job for Class2Table computes and assigns the properties of the Tables and the Columns created by the corresponing matcher.

- a *match* job for DataTypeAttribute2Column looks for elements of type Attribute that satisfy the condition at line 19, and for each one it instantiates an empty Column.
- an *apply* job for DataTypeAttribute2Column computes and assigns the properties of the Columns created by the corresponding matcher.

**Data parallelism.** In pure data-parallelism approaches, the input domain is partitioned and each task executes the same operations on a different partition. In model transformations terms, source and optionally target models are divided in submodels and each transformation task is responsible for transforming its assigned chunk. The approach in general reduces inter-thread communication, by eliminating shared data, and by concentrating collaboration issues in a final merging step of the generated partial results.

While we did not address this kind of parallelism in this paper we recognize its importance, especially when moving to distributed environments, where communication cost is higher. We plan in future work of studying this possibility and the interaction with task parallelism in model transformations.

### 3.2 Task Synchronization

A decomposition of a model transformation in parallel tasks may in general introduce synchronization issues for accessing shared data structures. Fig. 2 visualizing this problem by representing the Parallel Transformation, a read-only source model, and a set of read-write data structures, that comprise the Target Model, a set of Trace Links to store information about rule executions, and other generic runtime data structures used by the transformation code, or the transformation execution algorithm.

Dashed ellipses in Fig. 2 represent possible synchronization issues:

- the source model is read-only, hence concurrent reads do not require a synchronization mechanism;
- CRUD operations on the target model may require synchronization. This is true for CRUDs on model elements, or on single properties (for the sake of the discussion in Fig. 2 we distinguish operations on single-valued properties from operations on multi-valued properties). Moreover operations on properties may need to be synchronized with operations on elements. E.g., one thread may need to finish creating an element, before another thread tries to set a contained property.
- CRUD operations on trace links may require synchronization. For instance trace links may be stored in a collection that does not allow link creations to interleave. Moreover CRUDs on trace links need to be synchronized with CRUDs on model elements. E.g., a target element creation needs to be complete before the corresponding trace link can be connected to the element.
- finally the engine may require synchronization on other runtime data, coming from the transformation code or the internal engine implementation.

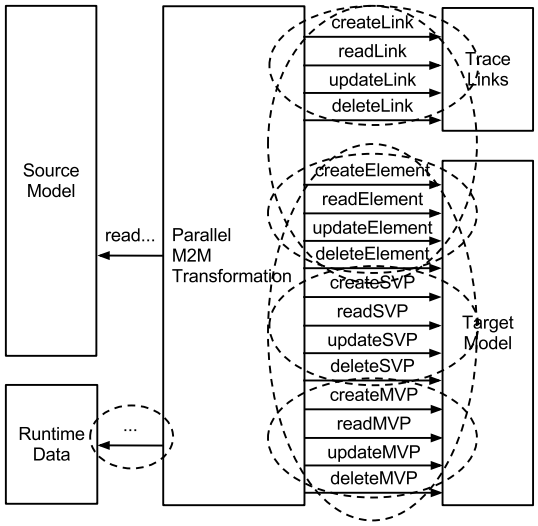


**Fig. 2.** Synchronization for concurrent data access in parallel transformations.

All these possible synchronization points make parallelization a difficult task for model transformations in a general purpose language and risk to hamper the actual gain in scalability. In the next section we will try to reduce the synchronization points by exploiting the specific constraints of ATL.

**One task per ATL rule.** When the transformation is written in ATL and we decide to assign a different task to each ATL rule, the number of operations that may require synchronization results strongly reduced w.r.t. the general case, as shown in Fig. 3:

- On the target model:
  - After they are created, model elements are not further modified by the ATL engine (but their properties are). However, element creations need to be synchronized, since they operate on the collections used by the EMF framework, that does not offer any support to concurrent access.
  - Single-valued properties are only created and contextually associated to a value. Since they are stored in EMF as Java references and they are not subject to modifications, they do not need synchronization.
  - Multi-valued properties are created and updated by adding elements during the transformation. EMF stores multi-valued properties in Java Lists that do not support concurrent update.
  - No synchronization is needed between operations on properties and operations on elements, since in ATL a property can be assigned only by the rule that creates the element. Hence, element creation and property assignment always happen in the same thread.
- On trace links:

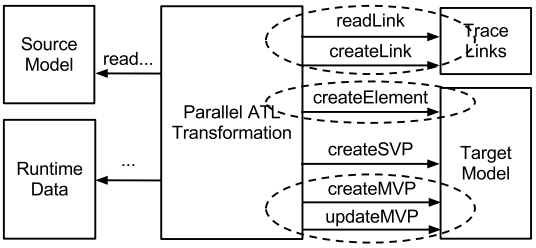


**Fig. 3.** Synchronization in parallel ATL transformations

- Trace links can be created by some rule, and only read by other rules, requiring a synchronization method. However no update or deletion is allowed.
- No synchronization is needed between element creation and link creation, since they always happen in the same rule/thread.

– On runtime data:
- ATL supports the definition of runtime data structures in the OCL part, but being OCL side-effect free they do not require synchronization.
- The engine does not introduce internal data structures that need synchronization.

While the synchronization in ATL results simplified w.r.t. Fig. 2, it still may represent a significant overhead. Critical sections for each one of the operations are short, since they consist in a single elementary operation on a collection, but the number of passages by a critical section is proportional to model size.

In the next section we will try to optimize the synchronization of operations over trace links. Instead, unfortunately, not much can be optimized about operations on the target model, since the creation of elements and properties cannot be avoided and EMF takes charge of this creation by using standard collections. An optimization in this sense should be done on the modeling framework side: EMF may provide non-blocking operations for element and property creation.

**One task per match/apply.** We can strongly reduce lock contention on accessing trace links by dividing each rule in two separate threads, as explained in Section 3.1, and organizing job dispatching in two phases (Fig. 4).

In a first *matching phase* all the matching jobs are executed. The jobs are responsible for creating elements and creating trace links and these two operations still need synchronization, since they involve concurrent matchers modifying the same collections.

A second *apply phase* is activated when all the matchers have completed execution. The *launch* message in Fig. 4 represents a hard synchronization point, and it negatively affects the total computation time especially when matchers have very different execution time. In the worst case of one matcher much slower than the others, the hard barrier prevents other threads to be launched and the

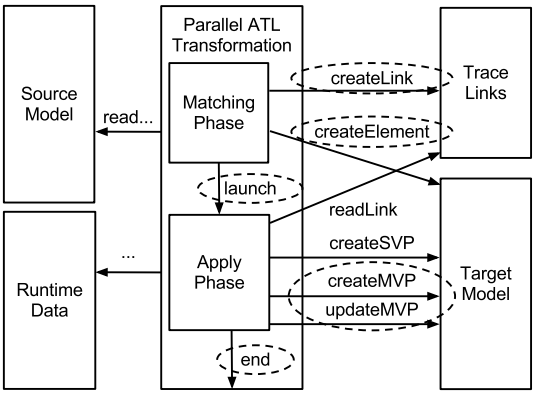


**Fig. 4.** Synchronization in two-phases parallel ATL transformations

other cores stay idle. An analogous behavior happens at the end of the transformation (but it is common to the previous approaches): the transformation termination is determined by the end of the slowest parallel job.

The gain in this wait stands in the fact that the trace link list at this point is read-only and no synchronization is needed during all the apply phase. To have an idea of the impact of this change, we have to consider that an ATL transformation executes a readLink operation each time it fills a reference in the target element, if the reference involves more rules (the most common case). In other words the number of readLink operations is of the same order of magnitude of the number of references in the target model. In Section5 we will provide experimental evidence of the resulting speed-up.

## 4 The Parallel ATL Engine

We have adapted the ATL compiler and virtual machine (VM) to implement automatic parallelization. The new engine has full support for declarative ATL and parallelizes computation based on matches rules, following the approach described in the previous section.

One of the criteria for the engine adaptation is to maintain the separation of concerns between compiler and VM: 1) the VM provides basic primitives to enable parallelization and guarantees the absence of race conditions on concurrent access; 2) the compiler defines the parallel tasks by translating ATL rules in low-level primitives.

In this model, decomposition (Section 3.1) is implemented in the compiler together with the division in phases, and synchronization of concurrent access (Section 3.2) is implemented in the virtual machine. A consequence is that other transformation languages that can compile towards our virtual machine, can in principle implement their own parallelization mechanism by using our primitives.

### 4.1 Virtual Machine

The main virtual machine primitive to launch a parallel task is a new opcode in the ASM bytecode language: the opcode *FORK* spawns a new job containing the operation passed as operand. The new primitive is analogous to the pre-existing *CALL* that launches another operation in the same thread. The *CALL* opcode takes an operation reference as operand and derives the program control in order to execute the opcodes the operation contains. When the processing of the opcodes terminates, control is returned to the caller operation that continues its execution. Conversely, the implementation of the *FORK* opcode wraps the operation referenced as its operand in a Java *Runnable*. Then, it calls a job executor to add this *Runnable* to the list of tasks to be launched. After this, the program control is immediately returned to the caller operation. A Java *ExecutorService* allocates the transformation tasks to a fixed number of threads passed as parameter at the transformation launch.

For thread coordination, the VM adds to the native library another data type, the integer semaphore, mapped to a Java Semaphore. In this way the compiler can instantiate semaphores and call primitives for initializing it, acquiring and releasing tokens. Token acquisition blocks the caller until a token becomes available (i.e. the integer semaphore contains a value >= 0). Integer semaphores are used by the compiler to synchronize threads at the end of each phase.

Finally, as discussed in Section 3.2 a set of operations on trace links and target models need to be made thread safe. In the VM implementation, a synchronized block is added to the VM operations for creating new model elements and updating multivalued properties in the target model. The same approach is followed for the operation in charge of adding new links to the hash registry of transformation tracelinks.

### 4.2 Compiler.

The default ATL compiler has been subject to the minimal modifications necessary to implement our parallelization algorithm. With respect to the old one, the new compiler:

- Adds to the beginning of the transformation the initialization of two integer semaphores with a negative number of tokens, equal to the number of rules.
- Creates a *match operation* for each rule and calls them sequentially using the *FORK* opcode.
- Creates a single *applyPhase operation* and calls it using a *FORK*.
- Adds as first instruction of the *applyPhase operation* a request for a token from the first semaphore. The request is refused until enough tasks have released a token on the semaphore.
- Adds as last instruction of each *match operation* a release instruction on the first semaphore.
- Creates in the *applyPhase operation*, an *apply operation* for each rule and calls them sequentially with *FORK*.

- Adds as last instruction of the main task a request for a token on the second semaphore.
- Adds as last instruction of each *apply operation* a release instruction on the second semaphore.

## 5 Experimental Evaluation

In this section we assess the performance of the parallel ATL engine by running two experimentations.

In the first experimentation we compare three equivalent implementations of the simple Class2Relational transformation, part of which has been presented in Listing 1.1: a Java implementation using the EMF Java API, an ATL implementation running on the standard ATL engine and the same ATL implementation running on the parallel ATL engine. The purpose is comparing execution time of parallel ATL over normal ATL and Java in a typical system. The ATL implementations have been developed by simplifying the Class2Relational transformation from the ATL Transformation Zoo [1]. For developing the Java implementation we provided an EMF expert with the specification of the ATL transformation.

We execute the three transformations feeding them with large class models generated by a stochastic metamodel instantiator that we developed adapting a publicly available tool from Obeo[5]. The tool allows us to define probability distributions for all the element types and properties of the metamodel and use them to drive instantiation. In our experimentation we define a single uniform probability distribution, and we use it for the number of Packages to generate, the number of Classes of each Package, the number of Attributes for each Class.

We use the instantiator to generate two sets of 10 models. For the first set the distribution is designed to produce models with an average of 10,000 elements, for the second the average is 1,000,000 elements. Given one of the two sets, and one of the three implementations, we produce 100 observations by running 10 times the transformation of the 10 models in the set. We summarize the results in Fig. 5 where each box represents 100 observations. The leftmost plot refers to models of 10,000 elements, the rightmost to models of 1,000,000 elements.

The tests have been performed on an environment with the following characteristics: 8-cores processor Intel Core i7-2760QM CPU @ 2.40GHz, with 8GB of physical memory, and running Ubuntu Linux (64 bits) version 12.10 (quantal) with Linux kernel 3.5.0-25-generic. As application environment, tests where performed on the Eclipse Platform version 4.2.1 on top of the OpenJDK Java Virtual Machine version 1.7.0_15. Note that the i7 CPU has only 4 physical cores, while it presents 8 cores to the OS by using hyper-threading.

In measurement, model loading and model serialization times were not taken into account. Loading and serialization are time-consuming tasks that also impact scalability when working with very big models. However, dealing with such problems lies out of the scope of the present work.

[5] https://github.com/Obeo/emf.specimen

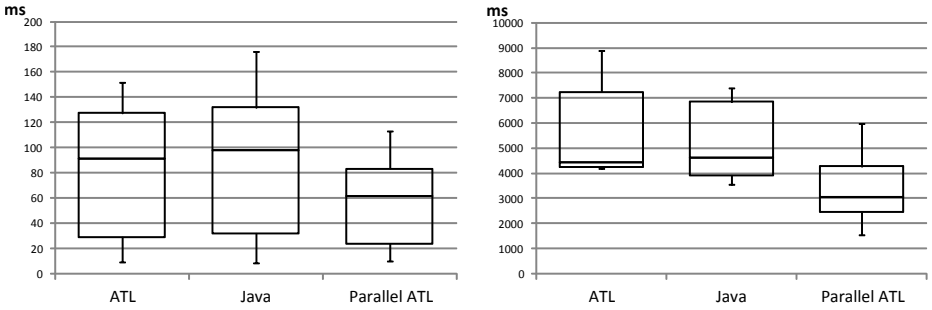


**Fig. 5.** Box plots summarizing the execution times of the same transformation on 4 cores. The leftmost plot refers to models of 10,000 elements, the rightmost to models of 1,000,000 elements.

From Fig. 5 we can observe that on the multi-core processor the ATL transformation on the parallel engine performs significantly better then the Java and standard ATL versions. The performance improvement is independent on the size of the models, resulting in a speed-up respectively of 1.49 and 1.48 (w.r.t. normal ATL) for the 10,000 and 1,000,000 elements model sets. We can see also that values have a lower dispersion with the parallel approach, that appears more robust (for 1,000,000 elements, the standard deviation is 1990ms for standard ATL, 2053ms for Java, and 1319ms for parallel ATL). While in this experiment we limited the processor to use always 4 cores, in the second experimentation we intend to measure the speed-up of the approach w.r.t. the number of cores.

Fig. 6 presents for each number of cores the correspondent speed-up of the parallel ATL engine over the default one. Tests are run in the same configuration as above but to stress the system and obtain more significant values we use a more complete and computationally expensive version of the Class2Relational transformation. While the original ATL implementation of Class2Relational counts only 6 rules, in order to keep all 8 virtual cores occupied during the transformation computation we have developed a new version that includes rules to translate super-classes, abstract classes and their attributes. The complete implementation is available in the paper's website.

To better evaluate the effect of model size on the speed-up, we run the test over 4 model sets of 10 models each. Model sets are generated as above and their respective average model size is 1000, 10000, 100000 and one million elements. Table 1 shows the average execution times in the different configurations.

The test shows a higher speed-up w.r.t. the previous experimentation, proving that the second version of Class2Relational is more parallelizable than the first. The speed-up curve, apart from small fluctuations shows a constant increase exhibiting good scalability on the number of cores. On the other side the graph shows that the speed-up is negatively influenced by model size, at least for small models. Parallelism on models of 1,000 elements perform significantly better than on models of 10,000 elements. However from a certain size the speed-up stays

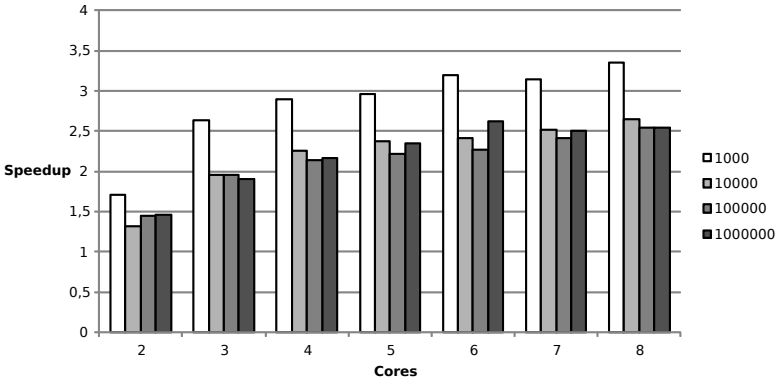


**Fig. 6.** Speed Up

constant and does not deteriorate even for VLMs. The speed-up over VLMs is more than 2.5, from which we can derive that the transformation over the parallel engine contains at least a 60% of parallelized code. The fact that only the first 4 cores are physical explains the little gain in speedup obtained from 4 cores (2.2) to 8 cores (2.5).

## 6 Related Work

Automatic parallelization is an under-studied subject in model transformations, and the parallelization properties of the most popular model transformation languages (QVT, ATL, ETL, Kermeta, ...) are still unknown. Conversely parallelization is a deeply studied subject in the graph-transformation community. A seminal work is [7], where the authors describe the concept of amalgamation as a generalization of the theorem of parallel graph transformation. Basically common parts of rule derivations are joined (amalgamated) in a single derivation. This allows to relax the requirement of parallel independence so that rules do not need to be independent anymore as long as the common part is amalgamated and executed first. [12] studies parallel independence in hierarchical graph transformations. The problem of parallelizing graph transformations is

**Table 1.** Average execution times (in milliseconds) and speed-up (between parenthesis) per model size and number of cores.

| # Elem. | Std. ATL | Parallel ATL | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 2 cores | 3 cores | 4 cores | 5 cores | 6 cores | 7 cores | 8 cores |
| 1000 | 83.4 | 48.9(1.7) | 31.7(2.6) | 28.8 (2.8) | 28.2(2.9) | 26.1(3.1) | 26.5(3.1) | 24.9(3.3) |
| 10000 | 1338.8 | 1013.4(1.3) | 685(1.9) | 592.8(2.2) | 565(2.3) | 556.2(2.4) | 532(2.5) | 505.3(2.5) |
| 100000 | 34942 | 24137(1.4) | 17849(1.9) | 16312(2.1) | 15742(2.2) | 15367(2.2) | 14483(2.4) | 13732(2.5) |
| 1000000 | 308290 | 211032(1.4) | 162487(1.9) | 142262(2.1) | 131393(2.3) | 117439(2.6) | 123122(2.5) | 121341(2.5) |

however very different (and more difficult) than parallelizing model transformations, especially because of need to handle rule recursion and convergence. Traditionally the attention of graph transformations is more directed on optimizing the matching phase, that is an NP task and the typical bottleneck for transformation performances.

In [13] the authors implement parallel graph transformations on multicore system with the VMTS tool. Similarly to our approach, the authors divide execution into two phases. The matching phase is parallelized but the apply phase is executed sequentially. In [15] the same authors distinguisg between transformation-level (i.e., paralellizing rules) and rule-level (i.e. parallelizing matching) parallelism and contribute an algorithm for the latter. but they do not provide concrete estimations of performance gain and scalability. Our approach would mostly be located at the transformation-level. Some authors apply parallel graph transformation to manipulate EMF models. The authors in [11] study parallel graph transformations on EMF based on the concept of amalgamated graph transformations. Viatra 2 [22] can rewrite multiple matches of a rewriting rule in parallel, but it does not support parallelization among different rules.

A related work on ATL is [9], where authors propose some early research ideas about parallelizing and distributing the language. They propose a distributed implementation based on MapReduce [10]. MapReduce is a programming model for expressing distributed computations on massive amounts of data and an execution framework for large-scale data processing on clusters of commodity servers. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. Our work may help in distributing ATL with the aim of making it a transformation language for distributed computation.

The work in [19] for parallelizing the XSLT language has many analogies to ours. The authors study the implicit parallelism of XSLT transformations, with different execution models, task and data parallelism, and they provide an engine implementation designed to work on multicore systems. They also show in a performance evaluation the speedup and scalability gains.

## 7 Conclusions and Future Work

In this article we added an implicit parallelization mechanism to the ATL language, and we studied how the structure of the language helps in overcoming some typical synchronization problems of parallelization. We believe that automatic parallelization may result beneficial to the general acceptance of MDE in industries handling big data.

While in this work we focused on task parallelism, in future we plan to study data parallelism for ATL and possible intermediate approaches. A study on static analysis of rule dependencies may help in anticipating some rule applications, without the need to re-introduce locks on data access. Finally implicit parallelism has its own drawbacks (difficulties in debugging, reduced control by the programmer over the parallel execution) that still need to be studied for ATL.

## References


1. ATL Transformation ZOO. `http://www.eclipse.org/atl/atlTransformations/`.
2. AUTOSAR Consortium: The AUTOSAR Standard. `http://www.autosar.org/`.
3. GraBaTs 2009 Case Study. `http://www.emn.fr/z-info/atlanmod/index.php/GraBaTs_2009_Case_Study`.
4. G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös. Incremental evaluation of model queries over emf models. In *Model Driven Engineering Languages and Systems*, pages 76–90. Springer, 2010.
5. J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development at the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications.*, Oct. 2004.
6. G. Blair, N. Bencomo, and R. B. France. Models@ run. time. *Computer*, 42(10):22–27, 2009.
7. P. Boehm, H.-R. Fonio, and A. Habel. Amalgamation of graph transformations: a synchronization mechanism. *J. Comput. Syst. Sci.*, 34(2-3):377–408, June 1987.
8. S. Burmester, H. Giese, M. Hirsch, D. Schilling, and M. Tichy. The fujaba real-time tool suite: model-driven development of safety-critical, real-time systems. In *Proceedings of the 27th international conference on Software engineering*, pages 670–671. ACM, 2005.
9. C. Clasen, M. D. Del Fabro, M. Tisi, et al. Transforming very large models in the cloud: a research roadmap. In *First International Workshop on Model-Driven Engineering on and for the Cloud*, 2012.
10. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
11. H. Ehrig, U. Golas, G. Taentzer, C. Ermel, and E. Biermann. Parallel independence of amalgamated graph transformations applied to model transformation. *Graph transformations and modeldriven engineering*, pages 1–21, 2010.
12. A. Habel and B. Hoffmann. Parallel independence in hierarchical graph transformation. *Graph Transformations*, pages 207–210, 2004.
13. G. Imre and G. Mezei. Parallel graph transformations on multicore systems. *Multicore Software Engineering, Performance, and Tools*, pages 86–89, 2012.
14. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1):31–39, 2008.
15. G. Mezei, T. Levendovszky, T. Mészáros, and I. Madari. Towards truly parallel model transformations: A distributed pattern matching approach. In *EUROCON 2009, EUROCON'09. IEEE*, pages 403–410. IEEE, 2009.
16. D. A. Patterson and J. L. Hennessy. *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 2009.
17. R. Pohjonen, J.-P. Tolvanen, and M. Consulting. Automated production of family members: Lessons learned. In *Proceedings of the Second International Workshop on Product Line Engineering-The Early Steps: Planning, Modeling, and Managing (PLEES02)*, pages 49–57. Citeseer, 2002.
18. J. Steel, R. Drogemuller, and B. Toth. Model interoperability in building information modelling. *Software & Systems Modeling*, 11(1):99–109, 2012.
19. Y. Sun, T. Li, Q. Zhang, J. Yang, and S.-w. Liao. Parallel xml transformations on multi-core processors. In *e-Business Engineering, 2007. ICEBE 2007. IEEE International Conference on*, pages 701–708. IEEE, 2007.

20. W. M. Ulrich and P. Newcomb. *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. Morgan Kaufmann, 2010.
21. T. Vajk, Z. Dávid, M. Asztalos, G. Mezei, and T. Levendovszky. Runtime model validation with parallel object constraint language. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, page 7. ACM, 2011.
22. D. Varró and A. Balogh. The model transformation language of the viatra2 framework. *Science of Computer Programming*, 68(3):214–234, 2007.